INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

From MARTE to dynamically reconfigurable FPGAs : Introduction of a control extension in a model based design flow

Imran Rafiq Quadri — Samy Meftali — Jean-Luc Dekeyser

N° 6862

March 2009

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--6862--FR+ENG

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE
INRIA
centre de recherche LILLE - NORD EUROPE

From MARTE to dynamically reconfigurable FPGAs : Introduction of a control extension in a model based design flow

Imran Rafiq Quadri* , Samy Meftali† , Jean-Luc Dekeyser‡

Thème COM — Systèmes communicants
Équipes-Projets DaRT

Rapport de recherche n° 6862 — March 2009 — 39 pages

Abstract: System-on-Chip (SoC) can be considered as a particular case of embedded systems and has rapidly became a de-facto solution for implementing these complex systems. However, due to the continuous exponential rise in SoC's design complexity, there is a critical need to find new seamless methodologies and tools to handle the SoC co-design aspects. This paper addresses this issue and proposes a novel SoC co-design methodology based on Model Driven Engineering (MDE) and the MARTE (Modeling and Analysis of Real-Time and Embedded Systems) standard proposed by OMG (Object Management Group), in order to raise the design abstraction levels. Extensions of this standard have enabled us to move from high level specifications to execution platforms such as reconfigurable FPGAs; and allow to implement the notion of Partial Dynamic Reconfiguration supported by current FPGAs. The overall objective is to carry out system modeling at a high abstraction level expressed in UML (Unified Modeling Language); and afterwards, transform these high level models into detailed enriched lower level models in order to automatically generate the necessary code for final FPGA synthesis.

Key-words: Real-Time and Embedded Systems, SoC Co-design, FPGAs, Partial Dynamic Reconfiguration, ISP, Control, MDE, MARTE, UML

* *Imran.Quadri@lifl.fr*
† *Samy.Meftali@lifl.fr*
‡ *Jean-Luc.Dekeyser@lifl.fr*

Centre de recherche INRIA Lille – Nord Europe
Parc Scientifique de la Haute Borne
40, avenue Halley, 59650 Villeneuve d'Ascq
Téléphone : +33 3 59 57 78 00 — Télécopie : +33 3 59 57 78 50

Du MARTE à reconfiguration dynamique partielle en FPGAs: Introduction d'une extension du controle dans un flot de conception basée sur les modéles

Résumé : System-on-Chip (SoC) peut etre considérée comme un cas particulier de systèmes embarqués et est rapidement devenue la solution de mise en oeuvre ces sysètmes complexes. Toutefois, en raison de la complexité de conception de SoC, il existe un besoin de trouver de nouvelles méthodologies et d'outils transparente pour traiter la co-conception de aspects SoC. Ce document traite cette question et propose une nouvelle co-SoC basés sur la méthodologie de conception Model Driven Engineering (MDE) et MARTE (Modeling and Analysis of Real-Time and Embedded Systems) norme proposée par l'OMG (Object Management Group), afin de élever les niveaux d'abstraction de la conception. Les extensions de cette norme, nous ont permis de passer de spécifications de haut niveau à l'exécution, tels que des plates-formes reconfigurables FPGAs, et permettent de mettre en oeuvre de la notion de reconfiguration dynamique partielle des FPGAs. L'objectif global est de réaliser la modélisation de systèmes á un haut niveau d'abstraction en UML (Unified Modeling Language), et ensuite, de transformer ces modèles de haut niveau dans les modèles de niveau inférieur afin de générer automatiquement le code nécessaire pour la synthèse.

Mots-clés : Systèmes Embarqués, Co-conception SoC, FPGAs, Reconfiguration partielle, ISP, Control, MDE, MARTE, UML

1 Introduction

Since the early 2000s, System-on-Chip (SoC) has emerged as a new methodology for embedded systems design. In a SoC, the computing units (programmable processors, hardware functional units), memories, I/O devices, communication channels, etc.; are all integrated into a single chip. Moreover, multiple processors can be integrated into a SoC (Multiprocessor System-on-Chip, MPSoC) in which the communication can be achieved through Network on Chips (NoCs). These SoCs are generally dedicated to target application domains (such as multimedia video codecs, software-defined radio and radar/sonar detection systems) that require intensive computations. According to Moore's law, rapid evolution in hardware technology doubles the number of transistors in an Integrated Circuit (IC) nearly every two years. As the computational power increases, more functionalities are expected to be integrated into the system. As a result, more complex software applications and hardware architectures are integrated, leading to a *system complexity* issue which is one of the main hurdles facing SoC co-design. The fallout of this complexity is that the system design (particularly software design) does not evolve at the same pace as that of hardware due to issues such as development budget limitations, reduction of product life cycles and design time incrementation. This evolution of balance between production and design has become a critical issue and has finally led to the *productivity gap*. System reliability and verification are also the other issues related to SoC industry and are directly affected by the design complexity. An important challenge is to find efficient design methodologies that raise the design abstraction levels to reduce overall complexity, while effectively handling issues such as accurate expression of inherent system parallelism: such as application loops; and hierarchy.

Currently High Level Synthesis (HLS) (or Electronic System Level) is an established approach in SoC industry. This approach raises the design abstraction level to some degrees as compared to traditional hand written HDL (Hardware Description Languages) implementations. The gap between the high abstraction levels and the low abstraction levels is often bridged using one or several *Internal Representations* (IRs) [24]. The behavioral (algorithmic) description of the system is written in a high level language such as SystemC [43] or a similar language, and is then refined into a RTL (Register Transfer Level) implementation using HLS tools. An effective HLS flow and associated tools must be *flexible* to cope with the rapid hardware/software evolution; and *maintainable* by the tool designers. The underlying low level implementation details are hidden from users and their automatic generation reduces time to market and fabrication costs. However, usually the abstraction level of the HLS tools is not elevated enough to be totally independent from low level details. Normally, the set of concepts related to an IR are generally difficult to handle due to absence of formal definitions of key concepts and their relations. The text based nature of a system description also results in several disadvantages. Immediate recognition of system information such as related to hierarchy, data parallelism and dependencies is not possible; differentiation between different concepts is a daunting task in a textual description and makes modifications complex and time consuming.

Model Driven Engineering [45] (MDE) is an emerging domain and can be seen as a *High Level Design Flow* in order to resolve the issues related to SoC co-

design. MDE enables system level (application/architecture) modeling at a high specification level allowing several abstraction stages (i.e. IRs). Thus a system can be viewed globally or from a specific point of view of the system, allowing to separate the system model into parts according to relations between system concepts defined at different abstraction stages. This *Separation of Views* (SoV) allows a designer to focus on a domain aspect related to an abstraction stage thus permitting a transition from *solution space* to *problem space*. Using a graphical modeling language i.e. UML (Unified Modeling Language) for system description increases the system comprehensibility. This allows designers to provide high-level descriptions of the system that easily illustrate the internal concepts (task/data parallelism, data dependencies and hierarchy). These specifications can be *reused*, *modified* or *extended* due to their graphical nature. Finally MDE's model transformations allow to generate executable models (or executable code) from high level models bridging the gap between these models and execution platforms.

FPGAs (Field Programmable Gate Arrays) are considered an ideal solution for SoC implementation due to their reconfigurable nature. Designers can initially implement, and afterwards, reconfigure a complete SoC on FPGA for the required customized solution. Thus FPGAs offer a migration path for final ASIC (Application Specific Integrated Circuit) implementation. Modern state of the art FPGAs also possess the capability to change their functionality at *runtime*, known as Partial Dynamic Reconfiguration (or Partial Runtime Reconfiguration) [33] (PDR); and introduce the domain of *Dynamic Reconfigurable Computing*. PDR allows to modify specific regions of an FPGA on the fly, thus exhibiting the notion of a *virtual hardware* with the advantage of time-sharing the available hardware resources for executing multiple (mutually exclusive) tasks. PDR allows task swapping depending upon application needs, hardware limitations and Quality-of-Service (QoS) requirements (power consumption, performance, execution time etc.). Currently only Xilinx FPGAs fully support this feature.

MARTE [41] (Modeling and Analysis of Real-Time and Embedded Systems) is an industry standard of Object Management Group (OMG), dedicated to model-driven development of embedded systems. MARTE extends UML, allowing to model the features of software and hardware parts of a real-time embedded system and their relations, along with added extensions (for e.g. performance and scheduling analysis). Although rich in concepts, MARTE lacks a design flow to move from high level modeling to execution platforms.

Gaspard [16], [23] is a MDE based MARTE compliant SoC co-design framework dedicated specifically towards parallel hardware and software; and it allows to move from high level MARTE specifications to different execution platforms. It exploits the inherent *parallelism* included in repetitive constructions of hardware elements or regular constructions such as application loops. Gaspard also focuses on a limited application domain, that of *intensive signal processing* (ISP) applications.

The main contribution of this paper is to present a novel MDE based design methodology for implementing the aspects of Partial Dynamic Reconfiguration from an extended MARTE standard. This design flow successfully responds to the major issues, for both users and designers of a typical HLS flow. Applications are graphically specified at a high abstraction level with UML and factorized expressions of parallelism, multidimensional data arrays and powerful

constructs of data dependencies are managed thanks to the use of the MARTE standard profile. The design flow allows to specify part of the reconfigurable system at a high abstraction level: notably the reconfigurable region and the reconfiguration controller. Afterwards, using model to model transformations, the gap between high level specifications and low implementation details can be bridged to automatically generate the code required for the creation of bitstream(s) for final FPGA implementation.

The rest of this paper is organized as follows. An overview of MDE is provided in section 2 while section 3 summarizes our MARTE compliant GASPARD framework. Section 4 describes PDR concepts while section 5 gives a detailed explanation of the deployment extension in MARTE. Section 6 details the related works and Section 7 illustrates our methodology related to implementing PDR supported FPGAs. A case study is present in section 8 followed by future works and perspectives. Finally section 10 details the conclusion.

2 Model Driven Engineering

MDE revolves around three focal concepts: *Models*, *Metamodels* and *Model Transformations*. A model is an abstract representation of some reality and has two key elements: *concepts* and *relations*. Concepts represent "things" and relations are the "links" between these things in reality. A model can be observed from different abstract point of views (views in MDE). The abstraction mechanism avoids dealing with details and eases re-usability. A metamodel is a collection of concepts and relations for describing a model using a model description language; and defines syntax of a model. This relation is analogous to a text and its language grammar. Each model is said to *conform* to its metamodel at a higher definition level. A metamodel can be viewed as an IR in an HLS flow. Finally, MDE permits to separate the concerns in different models, allowing reutilization of these models and to keep them human readable.

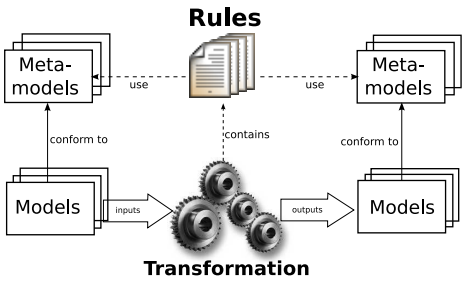

Figure 1: *An overview of Model Transformations*

The MDE development process starts from a high abstraction level and finishes at a targeted level, by flowing through intermediate levels of abstraction via *Model Transformations* (MTs) [51]; by which concrete results such as an executable model (or code) can be produced. MTs carry out refinements moving from high abstraction levels to low levels models and help to keep the different models synchronized. At each intermediate level, implementation details are added to the MTs. A MT as shown in figure 1 is a compilation process that

transforms a *source* model into a *target* model and allows to move from an abstract model to a more detailed model. Usually, the initial high level models contain only domain specific concepts, while technological concepts are introduced seamlessly in the intermediate levels. The source and target models each conform to their respective metamodels, thus respecting *exogenous* transformations [36]. A model transformation is based on a set of *rules* (either declarative or imperative) that help to identify concepts in a source metamodel in order to create enriched concepts in the target metamodel. New rules extend the compilation process and each rule can be independently modified; this separation helps to maintain the compilation process. The advantage of this approach is that it allows to define several model transformations from the same abstraction level but targeted to different lower levels, offering opportunities to target different technology platforms. The model transformations can be either unidirectional (only source model can be modified; targeted model is re-generated automatically) or bidirectional (targeted model is also modifiable, requiring the source model to be modified in a synchronized manner) in nature. In the second case, this could lead to a model synchronization issue [52]. For model transformations, OMG has proposed the Meta-Object Facility (MOF) standard for metamodel expression and Query/View/Transformation (QVT) [40] for transformation specifications.

3 GASPARD: MARTE compliant MDE based Co-Design Framework

Gaspard [16], [23] is a MDE oriented SoC co-design framework that utilizes a *subset* of the MARTE standard currently supported by SoC industry. In Gaspard as in MARTE, a clear *separation of concerns* exists between the hardware/software models, as shown in figure 2.

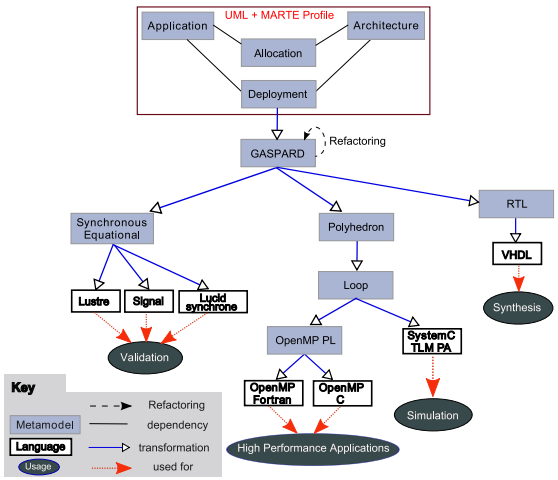

Figure 2: *GASPARD framework with deployment added at the MARTE specification level*

Gaspard has also contributed in the initial MARTE conception. One of the key MARTE packages, the *Repetitive Structure Modeling (RSM)* package has been inspired from Gaspard. Gaspard, and in turn RSM, is based on the Array-OL [9] model of computation (MoC) that describes the *potential parallelism* in a system; and is dedicated to intensive multidimensional signal processing (ISP). Array-OL itself is a specification language and not an execution model. In Gaspard, data are manipulated in the form of multidimensional arrays. The absence of limited number of dimensions in data arrays allows to represent data in a manner typical of their manipulation in ISP applications. For example, video processing applications handle two spatial and one temporal dimensions. Sonar chain is another kind of application, which handles spatial, temporal and frequency dimensions. RSM allows to models such applications.

RSM permits to describe the regularity of a system's structure (composed of repetitions of structural components interconnected in a regular connection pattern) and topology in a compact manner. Gaspard uses the RSM semantics to model large regular hardware architectures (such as multiprocessor architectures) and parallel applications. For an application functionality, both data parallelism and task parallelism can be expressed easily via RSM. A *repetitive* component expresses the data-parallelism in an application (in the form of sets of input and output *patterns* consumed and produced by the repetitions of the interior *part*). A *hierarchical* component contains several *parts*. It allows to define complex functionalities in a modular way and provides a structural aspect of the application: specifically, task parallelism can be described using such a component. The shape of a pattern is described according to a *Tiler* connector which describe the tiling of produced and consumed arrays. The *Reshape* connector allows to represent complex link topologies in which the elements of a multidimensional array are redistributed in another array. The difference between a *Reshape* and a *Tiler* is that the former is used for a connector that links two parts while the latter is used for a delegation connector: between a port of a component and ports of its parts. Another point to remember is that the ports (interfaces) of a component modeled in Gaspard have the MARTE *FlowPort* stereotype by default.

The MARTE Hardware Resource Model (HRM) concepts are inspired heavily from the preexisting hardware concepts in Gaspard. Finally the Generic Component Modeling (GCM) concepts are used as the basis for component modeling. Gaspard currently targets a limited application domain, namely *control and data flow oriented* ISP applications (such as multimedia video codes, high performance applications, anti-collision radar detection applications). The applications targeted in Gaspard are widely encountered in SoC domain and respect Array-OL semantics [9].

Gaspard also integrates the MARTE allocation mechanism (*Alloc* package) that permits to associate the applicative part of the system onto the available hardware resources (for e.g. mapping of a task or data onto a processor or a memory respectively). An example of an allocation is present in figure 3. The figure clearly illustrates the utilization of the MARTE concepts presented before. The RSM package represents the hardware repetitions and the application loops concisely in a declarative way, while the *Alloc* package allows to map the application on to the hardware resources.

Although MARTE is suitable for modeling purposes, it lacks the means to move from high level modeling specifications to execution platforms. Gaspard

bridges this gap and introduces additional concepts and semantics to fill this requirement for SoC co-design.

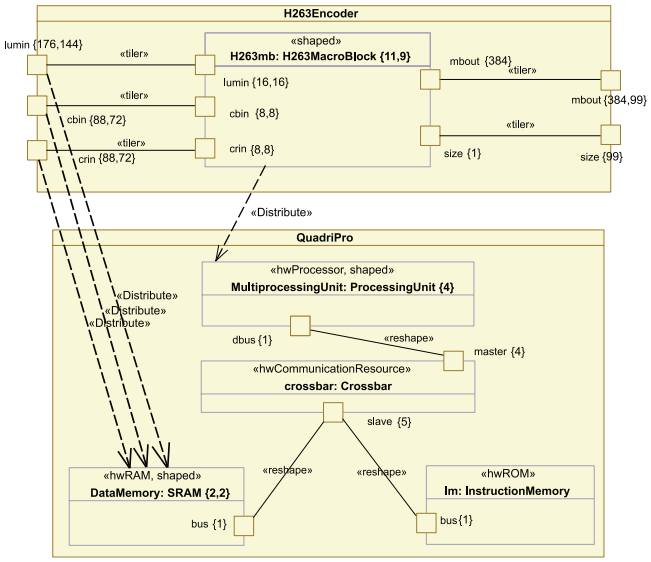

Figure 3: *An Allocation: mapping a part of a H.263 codec onto a QuadriPro architecture*

Gaspard also defines a notion of a *Deployment* specification level [2] in order to generate compilable code from a SoC model. This level is related to the specification of *elementary* components (ECs): basic building blocks of all other components having atomic functions. Although the notion of deployment is present in UML, the SoC design has special needs, not fulfilled by this notion. Hence, Gaspard extends the MARTE profile to allow deploying of ECs. To transform the high abstraction level models to concrete code, detailed information must be provided. The deployment level associates every EC (of both the hardware and the application) to an implementation (code) hence facilitating Intellectual Property (IP) reuse. Each EC ideally can have several implementations: e.g. an application functionality can either be optimized for a processor (written in C/C++) or written in hardware (HDL) for implementation as an hardware accelerator. Hence this level is able to differentiate between the hardware and software functionalities; and allows to move from platform independent high level models to platform dependent models for eventual implementation. Deployment provides IP information to model transformations to form a compilation chain in order to transform the high abstraction level models (application, architecture and allocation) for different domains: formal verification, simulation, high performance computing or synthesis. Hence deployment can be seen a potential extension of the MARTE standard to allow a complete flow from model conception to automatic code generation. It should be noted that the different transformation chains: simulation, synthesis, verification etc., are currently unidirectional in nature.

Once Gaspard models are specified in a graphical environment, model transformations are carried out via a transformation tool. However, since the stan-

dardization of QVT, few of the investigated tools are powerful enough to execute large complex transformations such as present in the Gaspard framework. Also none of these engines is fully compliant with the QVT standard. An alternative solution to QVT is the Eclipse Modeling Framework or EMF [18], that allows to create and modify models.

In order to solve this dilemma, In 2006, an initial transformation tool called MOMOTE (MOdel to MOdel Transformation Engine) was developed internally in the team that was based on EMFT QUERY [19]. MOMOTE is an enhanced Java framework that allows to perform model to model transformations. It is composed of an API and an engine. It takes source models as input and produces target models with each conforming to some metamodel. Another advantage of MOMOTE over the then existing transformation tools was that it supported external black box calls: e.g. native function calls, rule inheritance, recursive rule call and integration of imperative code. However, since that time, new tools such as QVTO and smartQVT have emerged that implement the QVT Operational language and are effective for handling the Gaspard model transformations. Currently, in order to standardize the model transformations and to render them compatible with the future versions of the MARTE standard; we have chosen QVTO as the future transformation tool for Gaspard. Current all the existing MOMOTE based transformation rules for each execution platform are being converted into QVTO based transformation rules.

MOCODE (MOdels to CODe Engine) is another internal Gaspard integrated tool that allows automatic code generation and is based on EMF JET (Java Emitter Templates) [20]. JET is a generic template engine for code generation purposes. The JET templates are specified by using a JSP (JavaServer Pages) like syntax and are used to generate Java implementation classes. Finally these classes can be invoked to generate user customized source code, such as Structured Query Language (SQL), eXtensible Markup Language (XML), Java source code or any other user specified syntax. MOCODE offers an API that reads input models, and also an engine that recursively takes elements from input models and executes a corresponding JET Java implementation class on them.

4 Basic PDR related concepts

Currently PDR is only fully supported by Xilinx FPGAs. Xilinx initially proposed two methodologies (difference based and module based) [54]; [55] followed by the *Early Access Partial Reconfiguration* (EAPR) flow [56]. The flow supports static nets in reconfigurable regions; and 2D reconfigurable modules, thus resolving the drawbacks present in the earlier modular design methodology. Part(s) of the FPGA remains static, while another part(s) is dynamically reconfigurable at run-time. *Bus macros* which are relationally placed macros (RPMs), are used to ensure proper communication routing between the static and dynamic parts during and after reconfiguration. Being CLB based in nature, they provide a unidirectional 8-bit data transfer. For Xilinx Virtex-II to Virtex-IV series FPGAs, Xilinx provides bus macros constructed of 2 CLBs. Bus macros are placed in a fashion that one CLB is placed inside the reconfigurable region while the other is outside in the static region. For modern state of the art Virtex-V FPGAs, single CLB based bus macros are also available [58].

At the heart of the PDR mechanism lies the *Internal Reconfiguration Access Port (ICAP)* [7], which is a subset of the SelectMAP Interface. It is an integral component that permits to modify the FPGA configuration memory at run-time. The configuration memory of FPGA contains the application specific data. Writing into a configuration memory is accomplished via configuration files known as *bitstreams* (that contain packets of configuration control information as well as the configuration data). The *granularity* of reconfiguration is also of importance. In the Virtex-II and Virtex-II Pro series FPGAs, the smallest unit of reconfiguration granularity is a frame. The number of bits present in a frame is directly proportional to the height of the device that is measured in CLBs. For example, for Virtex-II series FPGAs, the number of bits per frame ranges from 832 (the smallest device) to 9152 (the largest device) bits per frame. In the Virtex-IV series FPGAs, the smallest unit of reconfiguration granularity is a bit-wide column corresponding to 16 CLBs (or multiples, and this unit is independent of the different device sizes or families). A configuration frame of a Virtex-IV series FPGA contains forty one 32-bit words (1,312 bits per frame). The smaller granularity size allows more than one dynamically reconfigurable module to be placed vertically in the same region of FPGA. This is not possible in the earlier series FPGAs due to the frame based reconfiguration granularity.

The ICAP is present in nearly all Xilinx FPGAs ranging from the low cost Spartan-3A(N) to the high performance Virtex-V FPGAs [3]. For Virtex-II and Virtex-II Pro series, the ICAP furnishes 8-bit input/output data buses while with the Virtex-IV Series, the ICAP interface has been updated with 32-bit input/output data buses and the width can be alternated between 8 and 32 bits [14]. The ICAP cannot be directly connected to a system bus as it requires a controller to manage the data flow coming from the reconfiguration controller (RC). A classical ICAP controller is presented inside the OPB HwICAP core [62] (that allows interfacing with the ICAP). Another version is the PLB ICAP [13], however this paper focuses on the former version. The ICAP utilizes a mechanism of read-modify-write (RMW) [25] which allows a RC (a PowerPC or Microblaze) to modify the bitstream related to the reconfiguration module dynamically. After the modification, the modified bitstream is written back. The combination of the ICAP with the RC allows to build a self controlling dynamically reconfigurable system [7].

Virtex devices also support the feature of *glitchless dynamic reconfiguration*: If a configuration bit holds the same value before and after reconfiguration, the resource controlled by that bit does not experience any discontinuity in operation, with the exception of LUTRAMs and SRL16 primitives [33]. This limitation was removed in the Virtex-IV family. With the introduction of EAPR flow tools, this problem has also been resolved for Virtex-II/Pro FPGAs and designers do not have to explicitly exclude these resources from the reconfigurable module(s).

In this paper, respecting the terms as specified by Xilinx, a region of the FPGA to be reconfigured dynamically is termed as PRR or *Partial Reconfigurable Region*. A PRR can have several possible implementations or PRMs (Partial Reconfigurable Modules). An important point to consider is that all the PRMS of the PRR have the same external interface to ease compatibility. We now utilize these terms during the course of this paper. Figure 4 shows a general overview of a PDR system.

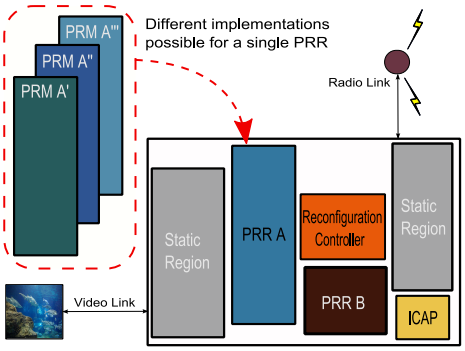

Figure 4: *An abstract illustration of a typical PRR system*

Usually an initial bitstream is loaded on to the FPGA which consists of the static portion as well as an initial PRM for the PRR(s). Afterwards, the controller only has to load the partial bitstream related to an alternate PRM for the same PRR. Using glitchless reconfiguration and the RMW mechanism, the *difference* between the two PRMS is noted and written back by the RC resulting in implementation of the new PRM.

5 Deployment level : a detailed overview

In order to generate an entire system from a high level specification, all implementation details of every EC have to be determined. Low level details are much better described by using usual programming languages instead of graphical UML models.

As explained before, the deployment level in Gaspard enables one to precise a specific implementation (IP) for each EC (of both application and architecture) among a set of possibilities. As compared to the deployment specified in [2], the deployment level has been modified to respect the semantics of traditional UML deployment.

The concept of *VirtualIP* has been introduced to express the behavior (functionality) of a given EC, independently from the compilation target. It links to all the possible implementations (IPs) for one EC. Finally, the concept of *CodeFile* is used to specify, for a given IP, the file corresponding to the source code and its required compilation options. The CodeFile thus identifies the physical path of the source code. It should be noted that the modeling of a CodeFile is not possible in the *UML composite structure diagram* but is carried out in the *UML Deployment diagram*. The desired IP is then selected by the SoC designer by linking it to the EC through the *implements* dependency.

Figures 5 and 6 show a clear description of the deployment level. The component *HuffmanCoding* is an elementary component of the Gaspard application (H.263 codec) present in figure 3. At the deployment level, this elementary component has several possible implementation choices. These choices can be for the same execution platform (same abstraction level) in a given language, or can be for different ones. In the illustrated example, the component can be implemented for simulation in SystemC or can be implemented as a hardware

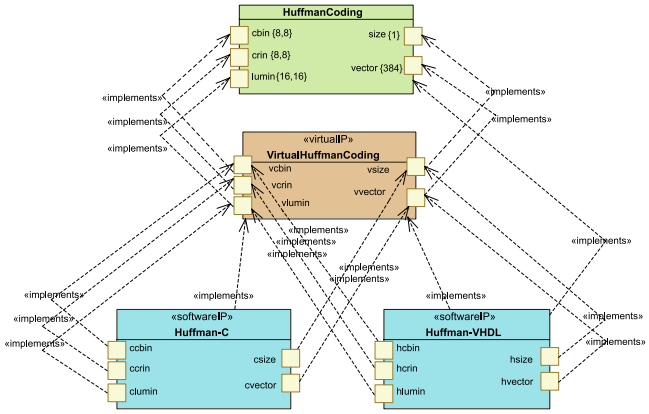

Figure 5: *Deployment of the HuffmanCoding elementary component*

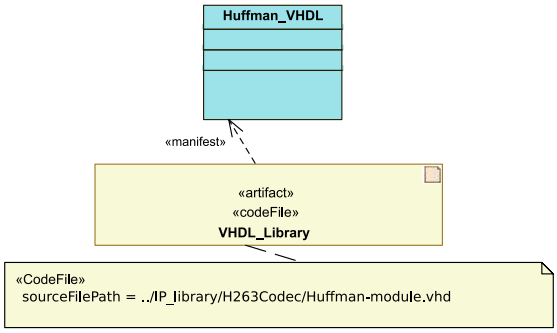

Figure 6: *The CodeFile artifact determines the physical path for the code related to an IP*

functionality: a hardware accelerator in an FPGA by synthesizable VHDL. The final *implements* dependency from the *Huffman-VHDL* component to the *HuffmanCoding* illustrate that this is the targeted implementation choice and the execution platform.

A fundamental limitation of the current deployment level is that for final compilation to an execution platform, an elementary component can be linked to *only one* implementation (IP). While this does make sense in regards to platforms where dynamic nature is not relevant, in the case of dynamically reconfigurable FPGAs, this is an important factor. We discuss our extended PDR supported Deployment level later in the paper.

6 Related Works

6.1 From UML to Synthesis

ROSES [12] is an environment for Multiprocessor SoC (MPSoC) design and specification, however it does not conform to MDE concepts and as compared to our framework; starts from a low level description equivalent to our deployment

level. [1] provides a simulink based graphical HW/SW co-design approach for MPSoC but the MDE concepts are absent. In contrast, [21] uses the MDE approach for the design of a Software-Defined Radio (SDR), but they do not utilize the MARTE standard as proposed by OMG and use only pure UML specifications. While works such as [15] and [35] are focused on generating VHDL from UML state machines, they fail to integrate the MDE concepts for HW/SW co-design and are not capable of managing complex ISP applications. MILAN [37] is another project for SoC co-design benefiting from the MDE concepts but is not compliant with MARTE. Only the approach defined in [31] and [32] comes close to our intended methodology by using the MDE concepts for SoC co-design. Yet the disadvantage is that in reality it only generates the ISP application which is implemented as a black box in a targeted FPGA; and there is no notion of a heterogeneous SoC and the MARTE and PDR aspects are absent. MOPCOM [29] integrates MDE and MARTE but is not oriented towards PDR. In [6], the authors present a design flow to manage partially reconfigurable regions of an FPGA automatically using SynDEx. A complete system (application/architecture) can be modeled and implemented, however the MDE concepts are strikingly absent. Similarly [8] present an HLS flow for PDR, yet it still starts from a lower abstraction level as compared to MDE.

6.2 Partial Dynamic Reconfiguration

In the domain of runtime reconfiguration, Xilinx initially proposed two design flows in [54] and [55] termed as the *Modular based* and *Difference based* approaches. The difference based approach is suitable for small changes in a bitstream but is inappropriate for a large dynamically reconfigurable module necessitating the use of the modular approach. However, both approaches were not very effective leading to new alternatives.

[50] presented a modular approach that was more effective than the initial Xilinx methodologies and were able to carry out 2D reconfiguration by placing hardware cores above each other. The layout (size and placement) of these cores was predetermined. They made use of reserved static routing in the reconfigurable modules which allowed signals from the base region to pass through the reconfigurable modules allowing communication between modules by using the principle of glitchless dynamic reconfiguration.

[4] implemented 1D modular reconfiguration using a horizontal slice based bus macro. All the reconfigurable modules that stretched vertically to the height of the device were connected with the bus macro for communication. They followed by providing 2D placement of modules of any rectangular size by using routing primitives that stretch vertically throughout the device [26]. A module could be attached to the primitive at any location, hence providing arbitrary placement of modules. The routing primitives are LUT based and need to be reconfigured at the region where they connect to the modules. A drawback of this approach is that the number of signals passing through the primitives are limited due to the utilization of LUTs. This approach has been further refined in [48].

In March of 2006, Xilinx introduced the *Early Access Partial Reconfiguration (EAPR)* [56] design flow along with the introduction of CLB based bus macros which are pre-routed IP cores. The concepts introduced in [50] and [4] were integrated in this flow. The restriction of full column modular PDR was

removed allowing reconfigurable modules of any arbitrary rectangular size to be created. The EAPR flow also allows signals from the static region(s) to cross through the partially reconfigurable region(s) without the use of bus macros. Using the principle of glitchless reconfiguration, no glitches will occur in signal routes as long as they are implemented identically in every reconfigurable module for a PRR. The only limitation of this approach is that all the partial bitstreams (PRMs) to be executed on a reconfigurable region (PRR) must be predetermined.

Works such as [3] and [44] focus on implementing softcore internal configuration ports on Xilinx FPGAs such as the pure Spartan-3 that do not have the hardware ICAP core rendering dynamic reconfiguration impossible via traditional means. In [44] a soft ICAP known as JCAP (based on the serial JTAG interface) is introduced for realizing PDR while [3] introduces the notion of a PCAP, based on the parallel SelectMAP interface, providing improved reconfiguration rates as compared to the JTAG approach. However this approach is only suitable to reconfigure very small regions of FPGA and since the design is not an embedded one, it is impossible to retrieve bitstreams from an external memory. This issue has been addressed in [10], where a complete reconfigurable embedded design on a Spartan-3 board has been implemented using a reconfigurable coprocessor. The user application can map to a number of potential coprocessors; and the reconfiguration controller can order the self reconfiguration of the system for the reconfigurable coprocessor, resulting in loading of the partial bitstream related to a potential coprocessor. The results show that this achieves a compromise between the works presented in [3] and [44].

In [13], a new framework is introduced for implementing PDR by the utilization of a PLB ICAP. The ICAP is connected to the PLB bus as a master peripheral with direct memory access (DMA) to a connected BRAM as compared to the traditional OPB based approach. This provides an increased throughput of about 20 percent by lowering the process load. [53] provides another flavor of a PDR architecture by attaching a Reconfigurable Hardware accelerator to a Microblaze Reconfiguration controller via a Fast Simplex Link (FSL) [60]. In [14], a customized ICAP controller is presented in order to speed up the reconfiguration process depending on a specific reconfiguration scenario. This controller can be implemented as either a PLB/OPB ICAP and offers the possibility of different memory implementations: slices or BRAMs.

Works such as [28] use ICAP to connect with Network on chip (NoC) to allow distributed access to speed up reconfiguration time. However the Read-modify-write (RMW) [7] mechanism is not supported which is an important factor to speed up reconfiguration time. This limitation has been resolved in [49] where an ICAP communicates with a NoC using a light weight RMW method.

6.3 Control in GASPARD

As explained before, Gaspard targets intensive signal processing applications. However in reality, the applications targeted in Gaspard are mainly data flow oriented and there is no notion of control. This is because the core formalism of data parallelism in Gaspard (its MoC or Array-OL) is based on *data dependency* descriptions and repetition operators; for expressing data parallel applications that compute large amounts of data in a *regular* manner. The behavior of these applications is completely fixed statically and cannot be changed at run-time.

Thus dynamic behavior is considered to be a disadvantage to the regularity and performance of Gaspard applications.

However, the field of reconfigurable computing is gaining a foothold in the SoC industry at an increasing pace. As dynamic behavior begins to appear more and more in these data ISP applications, e.g. mobile multimedia systems, due to market requirements, QoS etc.; its absence becomes a big constraint in current Gaspard applications. Thus suitable behavioral modeling concepts are required to address these issues.

An initial version of control for expressing dynamic behavior in Gaspard at the application level has been proposed in [30]. The control is state-based, as it is inspired from the synchronous mode automata (SMA) [34]. However unlike SMA, the control and data computations are specified in a separate manner allowing a clear distinction between the two. As a result, data computations can be specified independently from control. Extensions to this work have been proposed in [22]; [67] and address hierarchical and parallel compositions and formal semantics based on SMA and the Array-OL language. These works allow to express the control events as arrays not dissimilar to the data arrays (An infinite flow of control events is modeled as an infinite array) which also have data dependencies.

We now present a summary of the control modeling concept related to the applications targeted in Gaspard. A application task in Gaspard can have several *exclusive* running modes. The choice of the activated mode at run-time is determined by an automata that serves to control that task. Thus two distinct components can be distinguished: the State Machine Component (SMC) which is a controlling component and the Mode Switch Component (MSC) that corresponds to a mode switch between the different available modes of the task in question.

6.3.1 SMC and UML State machines

The SMC is associated with *UML state machines* [42] and reacts to external events. Determinism is also an important property for the state machines, i.e., for each state, input events lead to firing of at least one transition (either a self transition or a transition to another state). While traditional UML state machines are associated to a component: state machines react to some events issued in its context classifier, Gaspard state machines (or GSMs as we call them) are associated with a Gaspard component to give a precise external view with regards to its ports. A GSM is constrained to finish a transition before the arrival of a new value on the input ports of a SMC. To avoid ambiguity, we use this name for the remainder of the paper. The interfaces of the SMC are represented by *Ports* and stereotyped accordingly as *FlowPorts*. The shape of a port indicates the number of values arriving at a port at one instance of time. As this proposal takes only data flow into consideration, the shape corresponds to one reaction of the state machine and is always defined as {1}. The input ports of a SMC can be either *event* or *state* ports. Event ports serve as triggering a transition in the associated state machine and are normally of boolean type. The events used in Gaspard for triggering a transition is generally a *ChangeEvent* [42]. This event has a *changeExpression* which is a boolean expression that can result in a change event. The second type of event is the *AnyReceiveEvent* which is considered as a default event when all the triggers of the transition of a

state are not satisfied. They are expressed as the *all* statement in the modeling of the GSM.

Values associated to state ports are termed as *state values* and identify the different states in the GSMs. The input state port for a SMC indicate the initial state upon entering the GSM. For a hierarchical GSM, multiple initial states may be required. This issue is discussed in detail in [67]. A SMC also supports two kinds of output ports: *mode ports* and *state ports*. The GSMs associated to the SMC carries out transition functions on the states and each state is associated to one mode. The output mode port thus carry mode values to the MSC, which are determined by the transitions of the GSMs. The output state ports are similar to the input state ports and provide next state of the GSM (the next state after a transition). The mode values are defined in the application in an enumeration and are independent of SMC and its associated MSC. Figure 7 collectively shows the two UML diagrams related to an SMC and its associated state machine.

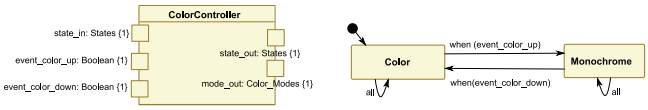

Figure 7: *An example of a State Machine Component (SMC) and associated GSM*

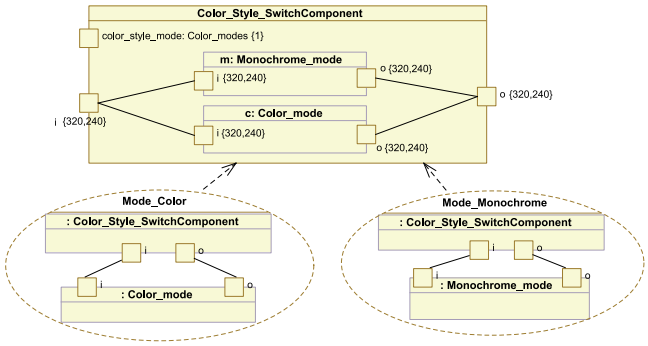

Figure 8: *An overview of MSC with data flow values on input/output ports*

6.3.2 MSC and UML Collobrations

The MSC is associated with *UML collaborations* [42] and serves to switch between the different exclusive modes present: It has at least one mode. A MSC has an input mode port which obtains the mode values from the SMC. It can also have input / output ports for expressing the data flow related to the relevant task. An MSC acts on the mode values and executes the corresponding mode. Only one mode can be selected at time t depending upon the mode value present in the input mode port at t. All the modes present in a MSC share the same interface (which is equivalent to the MSC interface except the mode port) and are connected to the MSC with the help of *delegation* connectors. While

the structure of the MSC can be defined using the MARTE general component concepts as defined in GCM, the execution semantics of an MSC and the internal collaborations of its internal parts are not evident. For this reason, UML collaborations are associated with a MSC. These collaborations specify roles of components (instance level collaboration) via usage of connectors and parts in composite structures. A collaboration specifies the relation between some collaborating components (or roles). Each of these roles provides a specific function, and executes some required functionality in a collective way. Only the concerned aspects of a role are included in a collaboration while others are omitted. Figure 8 shows an example of a MSC. The two collaborations depict the behavior of the MSC *Color_Style_SwitchComponent*. The name of the collaborations correspond to the mode values and thus these collaborations define the activity of the MSC upon receiving a particular mode value. For example, the collaboration *Mode_Color* shows the relationship between the MSC and the mode *Color_mode* (indicating that mode value *Mode_Color* switches the current executing mode to *Color_mode*). The connectors between them are shown in the collaboration. As in this mode only *Color_mode* is to be executed, the second mode or *Monochrome_mode* is omitted. The collaboration is finally linked to *Color_Style_SwitchComponent*.

6.3.3 Creation of a Gaspard Mode Automata

To create a Gaspard mode automata (GMA), first its internal structure: a composition of a SMC and a MSC, is constructed. The SMC produces mode values which are taken by the MSC, that executes a switch function between the modes present in the MSC. Compared to synchronous mode automata, the computations are not set in the states of a state machine, but are placed in the MSC. This composition is termed as a GMS (Gaspard Macro Structure). An abstract representation of GSM is also present in figure 9 and illustrates a complete Gaspard control structure. It is evident that this is not a UML diagram, due to the reason that a state machine (respectively a GSM) diagram cannot be modeled in a composite structure diagram (for modeling of Gaspard components). In order to shown the global view of a simple Gaspard control structure, we have include the state machines as well. While an abstract representation, this is very close to UML modeling. In order to simplify the illustration, *event_color_up* and *event_color_down* are only shown as a single event i_e.

Once the GSM is constructed, it is placed in a repetition context (RT). For SMCs with hierarchical GSMs, hierarchical RTs can be specified as defined in [67]. In a RT, a SMC can be executed in parallel (each repetition of SMC is independent of other SMCs; and the SMC has no memory of the previous state as the inputs required by each successive repetition of a SMC are only present at one time) or a sequential manner (a dependency exists between the repetitions of a SMC allowing to introduce the concept of memory of previous states). Currently in Gaspard we only address the second approach. A dependency in the sequential execution is called an *Interrepetition Dependency* or IRD. A *dependency vector* associated to the IRD expresses the dependencies between the repetitions inside the RT. If the depended repetition is not defined in the repetition space, a default value is selected. The *DefaultLink* provides default value for repetitions whose dependency for the input is absent. The GSM should be placed in a RT with at least one IRD specification, offering following advan-

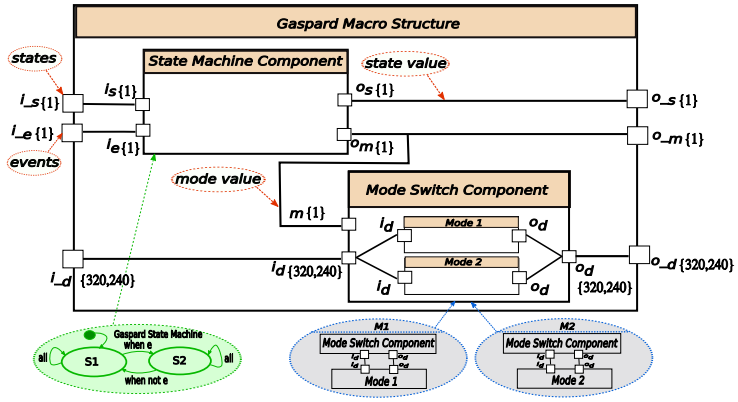

Figure 9: *Overview of the GMS*

tages.: A GSM only represents one transition function from a source state to a target state, where as a SMA has continuous transitions. Hence the GSM needs to be repeated for multiple transitions. An IRD permits a sequential execution, making it equivalent to the execution of a synchronous mode automata. Secondly, the RT is transformed into a serialized RT. This allows to construct a mode automata.

A global representation of the complete GMA structure is present in figure 10.

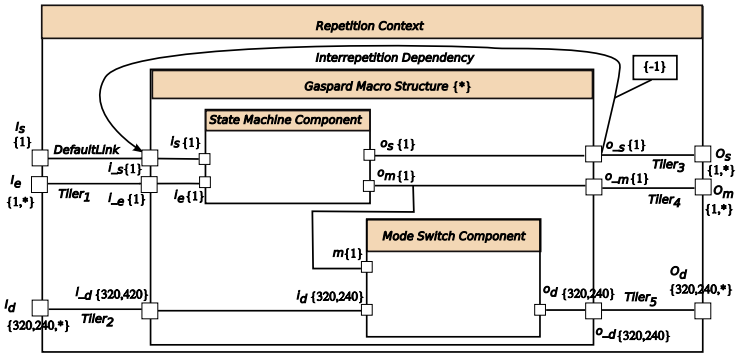

Figure 10: *An overview of Gaspard Mode Automata*

6.3.4 Summary of Related works

For our implementation purposes, we have focused mainly on the Xilinx EAPR flow methodology [33] as it is openly available and can be adapted to other PDR architecture implementations. While there are lots of related tools, works and projects; we have only detailed some and have not given an exhaustive summary. To the best of our knowledge, only our methodology takes into account the following domain spaces: SoC HW/SW co-design, ISP applications, control/data flow, MDE, MARTE standard and PDR; which is the novelty of our design flow.

7 Proposed Design Flow

Our design flow is inspired from the works present in [32] where a Gaspard application modeled at the UML level was successfully converted as a hardware functionality (while keeping the multidimensional arrays and repetitions specified at the modeling level). However this design flow only allows to create one configuration for final FPGA implementation using commercial tools and lacks the dynamic aspects required to implement PDR. Another drawback of this flow is that the final hardware design (the hardware accelerator) is implemented in an FPGA as a black box and there is no notion of a heterogeneous system (processors, buses, etc.) that communicate with this hardware accelerator. Also the MARTE modeling concepts are absent in the modeled applications.

As seen from the section related to PDR, a reconfiguration controller (RC) is necessary for dynamic reconfiguration of a PRR. Although only a state machine based controller can be used for reconfiguration, this cannot be an *internal dynamic self reconfiguration*. Usually in a processor based controller, a part of the code relates to interfacing with the ICAP core and fetching the necessary frames or bit wide columns (depending upon the FPGA series); while another part is the state machine which basically alternates between the different implementation choices. The first part can be viewed as an essential macro while the 2nd part depends upon the nature of the application, the number of PRRs, their corresponding PRMs and the mechanism to change the PRMs of a PRR. Either the RC acts upon the events provided by an external environment, or depends solely upon the tasks of the application present in the static/dynamic portions. Some parts of the application can be present in the static portion (even in the RC itself), while a part(s) can be dynamically reconfigurable [53].

Normally in PDR based systems (we avoid the discussion related to PDR based NoCs), we see a trend to change either the *tasks* of an application or the application itself. However in our design flow, we focus mainly on changing the implementations related to the ECs of an application. This offers two advantages. An application can retain the same *structure* and the same *functionality* while differing partly in the manner by which it is *implemented*. This implementation choice can arise due to several factors such as the available hardware resources, power consumption, reconfiguration time etc. The other advantage is that by changing the ECs, it is possible to partly change the functionality of the application.

In Figure 11 we present our MARTE based PDR design flow in order to implement the aspects of PDR. Initially the application is modeled via UML and MARTE concepts; and is independent from any implementation details. Afterwards, the UML2MARTE model transformation allows to transform the UML model into a MARTE model. This model corresponds to the MARTE metamodel. Afterwards this MARTE model is transformed into a Gaspard model by the MARTE2GASPARD transformation. Via GASPARD2RTL transformation, the RTL model is created which corresponds to a low abstraction level of an hardware accelerator (or several accelerators in the case of PDR) able to execute the initial modeled ISP application. The RTL model provides a nearly accurate estimation of the resources required for the resulting design implementation. An exploration process (not illustrated in the figure) is performed according to these estimations. Finally using MOCODE, it is possible to convert the models to source code. Once the source code for the application (implemented as

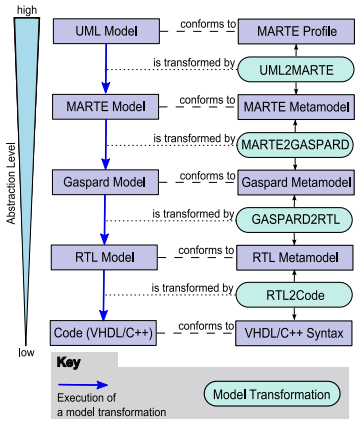

Figure 11: *The proposed model driven design flow for PDR*

a hardware accelerator) and the reconfigurable controller (state machine part) is obtained, the reconfigurable architecture can be created by using the Xilinx EDK tool [59]. The application implemented as a hardware design can be imported as a peripheral in the architecture while the code corresponding to the state machine can be used as part of the source code for the RC (a PowerPC in our case). Once the system has been created, usual synthesis flow can be invoked using commercial tools such as Xilinx ISE [61] and PlanAhead [17] for final implementation. A further extension of our work can be to model the whole PDR system (application and architecture) at the MARTE level for a complete automatic generation. We have proposed an initial modeling of the PDR architecture in [46] by introducing new concepts in MARTE HRM package, but the corresponding model transformations have not been carried out. Figure 12 shows a global overview of the modifications related to the Gaspard framework.

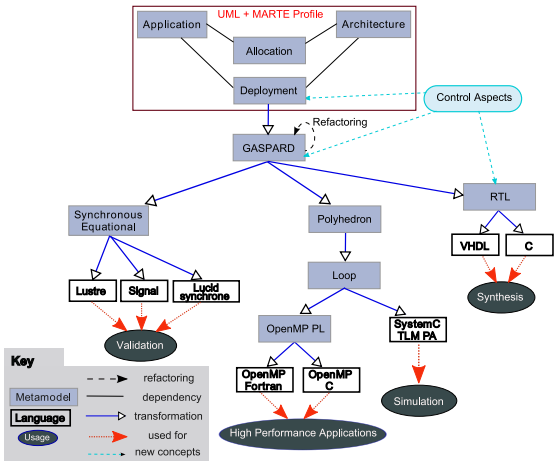

Figure 12: *Gaspard framework with added concepts for integration of PDR aspects*

Our aim is not to replace the commercial FPGA tools but to aid them in the conception of a system. While tools like ISE and PlanAhead are capable of estimating the configurable FPGA resources (CLBs and in turns the slices) required for implementing the hardware design, this resource estimation is only possible after initial synthesis. In our design flow, the ECs can be synthesized independently to calculate the consumed FPGA resources. This information can be then incorporated into the model transformations, making it possible to calculate the approximate number of consumed FPGA resources of the overall application (at the RTL model) before final code generation and eventual synthesis. Thus the designer is able to compare the resources consumed by the modeled application and the total resources available on the targeted FPGA resulting in an effective Design Space Exploration (DSE) strategy. If the application is too big to be placed on the FPGA, the designer can carry out a *refactoring* of the application. It should be noted that a refactored Gaspard application remains a Gaspard application.

As currently only Xilinx FPGAs support the features of PDR, our modeling methodology revolves around the Xilinx reconfiguration flow as it is openly available and flexible enough to be modified. While this does make the architectural aspects of our design flow restricted to Xilinx based technologies, it is an implementation choice as currently no other FPGA vendor supports this feature. It should be noted that our methodology can be used as a building block to support other non standard PDR implementations based on Xilinx FPGAs (use of Soft ICAP cores for example). Our contribution does not relates to creating a new PDR architecture methodology per se at the RTL level, but is based on how the PDR methodology can be raised to a higher abstraction level, to reduce design complexity and to create a generic PDR approach for implementing all ISP applications supporting our MoC. This approach can then be taken as an input by the designers who contribute to the optimization of the PDR aspects at the RTL level.

7.1 Initial implementation of PDR at RTL level

We first investigated the architecture related to implementing PDR in Xilinx FPGAs. Although PDR is a growing domain, we found that there is not sufficient initial information for a novice designer to implement this feature. Also the tools for implementing PDR are still in the development phase: for example, the EAPR flow is still not available for the Xilinx Virtex-V and the recently released Virtex-VI series FPGAs [65].

In Figure 13 we present the structure of our reconfigurable architecture that was implemented on a Xilinx Virtex-II Pro XC2VP30 FPGA on a XUP Board [66]. A Reconfiguration Controller (a PowerPC in this case) connects directly to the high speed 64-bit PLB bus and links with the slower slave peripherals (connected to the 32-bit OPB bus) via a PLB to OPB Bridge. The buses and the bridge are a part of the IBM Coreconnect technology [27]. The OPB bus is attached to several peripherals: A SystemACE controller for accessing the partial bitstreams placed in an external onboard Compact Flash (CF) card. A SDRAM controller for a DDR SDRAM present onboard, permits the partial bitstreams to be preloaded from the CF during initialization for decreasing the reconfiguration time. The ICAP core is also present in an OPB peripheral (Xilinx OPB HwIcap Core) and carries out partial reconfiguration using

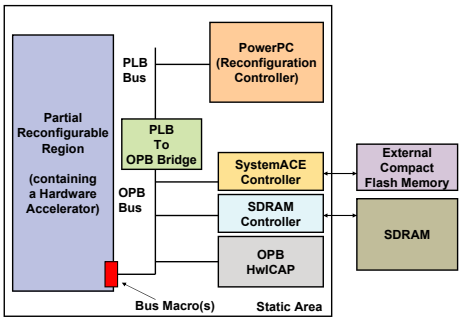

Figure 13: *Block Diagram of the architecture of our reconfigurable system*

the RMW mechanism. The static portion of the FPGA is connected to a Reconfigurable Hardware Accelerator (RHA) via bus macros. Although the RHA can be placed with the fast PLB bus, it is an implementation choice to connect it with the OPB bus. An internal memory can also be used to store the partial bitstreams depending upon the application size. Since in general, our targeted ISP applications cannot be placed inside the internal memory due to their sizes, an external memory was used. Initial experiments were carried out in order to understand the PDR mechanism. As explained previously, using the MDE design flow, we are mainly interested in generating the RHA part and the state machine part of the RC. The current model transformations are able to generate the synthesizable HDL code for a hardware accelerator for final static implementation, however they have to be modified for the PDR aspects. Another important point to consider is the integration of the PDR control in the Gaspard framework. Our proposition related to this issue is presented in the next section.

7.2 Modifications in existing Deployment level

As elaborated before, the notion of control has already been presented in Gaspard at the application level for the synchronous domain. However, in reality, this is currently only a theoretical approach and the model transformations are not present to implement this proposition.

In relation to PDR and specifically the concept of the RC, we were presented with a choice for implementing the control. While control in application allows the application to be independent of the targeted architecture, it requires multiple associations (of the application tasks onto the hardware resources) and multiple deployments. Control at only the architectural level allows the architecture to be independent of the application while suffering from the disadvantage of the absence of a dynamic hardware accelerator. Control at only the association level can be used to reduce the number of active executing units (hence decreasing the total power consumed) but offers no dynamic softcore or hardcore IP. Control at the deployment level renders both the application and the architecture reusable and the IPs (actual implementations) related to the elementary components can be alternated supporting IP reuse. In order to integrate the aspects of PDR, we extended the current mechanism of deployment by first introducing the notion of a *Configuration*.

7.2.1 Configurations

When viewed from an implementation point of view, as an EC can be linked to one final implementation (IP) among the different choices (if any); the final overall implementation of either the application or the architecture (or the mapping of the two forming the overall system) is a static one. We term this collective composition as a *Configuration*. The current model transformations for RTL chain only allow to generate one hardware accelerator (hence one configuration) for final FPGA implementation.

While control at the application level does allow a semi dynamic behavior at the application level in the synchronous domain, this is a *task level* dynamic nature and basically can modify the structure of the application (the different alternating modes for one task can be hierarchically composed in different manners). While the control at the deployment level allows *IP level* dynamic nature and safeguards the same structure for the application (or the architecture) by only changing the underlying functionality.

Our proposal allows to create several configurations for the final implementation(s). For this we create a new stereotype termed as a *ConfigurationInfo*. This concept can be added to the *implements* dependency which links an IP to the corresponding EC. This new stereotype has a *tagged value* (an attribute) termed as *ConfigurationNumber* which can be specified by the designer to determine that this particular implementation of the EC is related to which final configuration. It is thus possible for an EC to have the same IP in different configurations. This point is very relevant to the semantics related to partial bitstreams in PDR. By changing the metamodel and the corresponding model transformations related to the deployment level, it is now possible to link several IPs with an EC with each link specifying a specific configuration. We apply a condition that for any number of n configurations with each having m ECs, each EC of a configuration must have *at least* one IP.

Finally, the designer needs to add additional information by creating two enumerations, *Modes* and *States*. The first contains as entries, the IPs selected for all the possible configurations. The second contains the global states as entries. With information provided in deployment and the model transformation rules, each global state (configuration) is linked with its respective IPs. Each global state also has a boolean flag with a default value of 0. A value of 1 indicates that this global state is selected as the initial state/configuration for the GSM. This information is then passed onto the control concept modeled in the second phase of deployment using the model to model transformations. Figure 14 represents the extended version of the deployment related to an EC while omitting the enumerations; and figure 15 represents an abstract global overview of the deployment semantics.

By modifying the current RTL level model transformations, it is possible to generate different hardware accelerators (hence different configurations). Although a large part of the generated VHDL code (top level code and the code corresponding to the instantiated sub components) remains the same, the code corresponding to the EC is changed for each configuration. This choice of creating multiple hardware accelerators was chosen to remove ambiguity and to ease the creation of partial bitstreams. While it is possible to create only one hardware accelerator and change the low level EC implementations; and thus

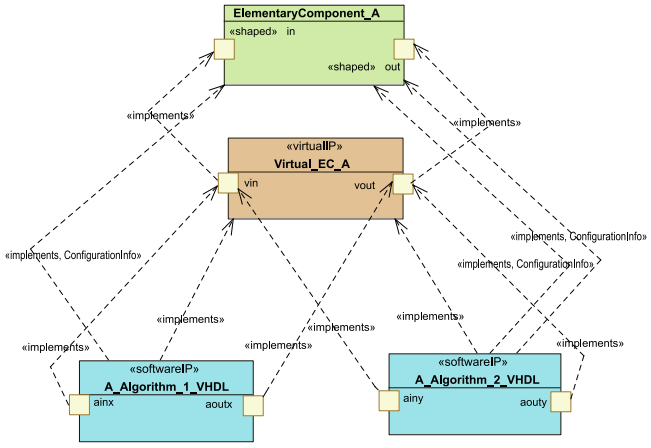

Figure 14: *Extended version of the deployment for an elementary component*

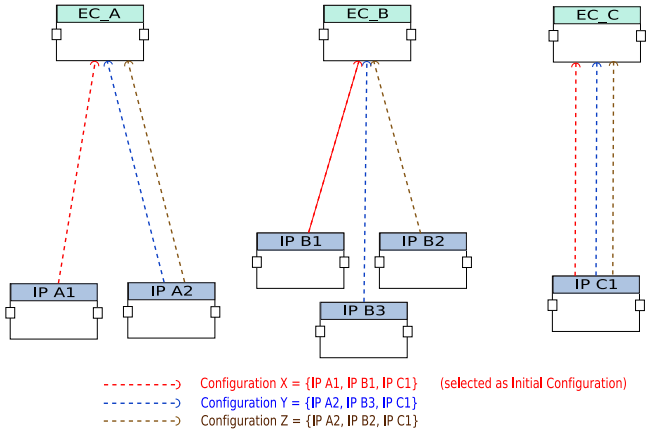

Figure 15: *Extended Deployment mechanism: for an application with three ECs*

create different configurations manually, it is a tedious task which augments in complexity depending upon an increase in the numbers of ECs or configurations.

Once the configurations are created, each one can be treated as a code for a partial bitstream (PRM) related to the RHA (PRR). Although this extension does allow to create different configurations, the RC in the FPGA is created manually for final PDR implementation. In order to automatically generate the state machine part of the RC, it is evident that this extension is not sufficient. We then turn to the existing control concepts present in [67] in order to solve this issue.

7.2.2 Modification of the existing Control concept

While the control presented in [67] is sufficient for introducing control flow at the application level, it is not compatible for integration at the deployment level. Hence we first carry out the necessary modifications. A bottom to top approach is adapted here to explain the control semantics; starting from the MSC.

In GMA, the different modes present in a MSC are modeled as *instances*. Each mode is an instance of a *component* (task) which can be either elementary, repetitive or hierarchical in nature. However modeling at the deployment level is only concerned with component themselves (The EC, the Virtual IP and the related IPs are all components). Hence the first change is that the modes (IPs) belonging to a MSC are modeled as components and not as instances.

The second change is related to the representation of the data flow entering and exiting the MSC (correspondingly the GMS, the RT and finally the GMA) itself. As the control is applied at the application (either at the global level of the application or at a sub level), the data flow entering and existing that level has to be expressed also (resulting in the input and output ports of the MSC, the GMS, the RT, the GMA; and the corresponding connectors). However this data flow is not expressed explicitly in the deployment level. The reason being that an IP for an EC essentially replaces that EC during final implementation, and the ports of the EC at the modeling level correspond to the ports of that IP via the *implements* dependency. Hence for all IPs belonging to an EC, their input/output data flow values are equivalent to that of the EC at an instance of time. For this reason, we suppress the input and output data ports of the MSC. Model transformations are capable to link the ports of each of the IPs in an MSC to the corresponding EC. This modified version of an MSC is termed as a *Deployed Mode Switch Component* or DMSC. It is evident that each DMSC is related to an EC and contains as modes, all the related available IPs.

The third change is related to the collaborations which express the behavior of a DMSC. As evident from the second modification, a collaboration only show the DMSC and corresponding mode. As compared to traditional MSC collaborations, the delegate connectors are absent. Figure 16 shows a DMSC and its collaborations. A limitation of the UML collaborations is the inability to model internal elements as components. For this reason, the IPs are modeled as instances and during model transformations it is possible to select the owner (the component itself) of the modeled instance. This approach has not been applied on the internal elements of the DMSC themselves (the IPs acting as modes) to avoid confusion relating to the modeling of the DMSC.

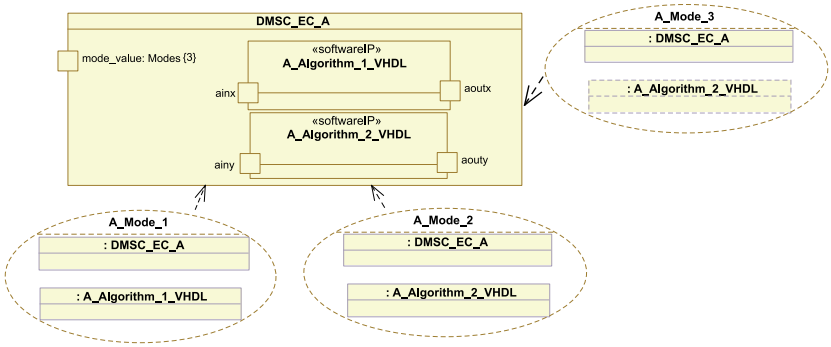

Figure 16: *The Deployed Mode Switch Component related to the elementary component EC_A*

An important point to remember is that the traditional macro structure or GMS as defined in the earlier proposition contains a SMC for controlling a single

MSC. However, in our proposition, as a DMSC is related to only one EC, and an application can have several ECs; this entails the creation of several DMSCs being controlled by a single SMC. This is one modification related to the overall control concept. This in turns produces the second modification related to the SMC.

The output mode port of a SMC in a GMA has a shape of {1}, indicating the only one mode value is produced at an instance of time; which is then treated as input for the single MSC. As we deal with multiple DMSCs (in turn a global GSM composed of several parallel sub-GSMs) with each requiring different mode values for the same instance of time t, we modify the shape value of the output mode port of the SMC. For n configurations, the value of the shape of the mode port is set to n as well. This modified SMC is termed as a DSMC or *Deployed State Machine Component*. This indicates that a n-sized array of mode values is produced. Correspondingly all DMSCs also have the same shape value on their corresponding input mode ports. Each DMSC receives the array of mode values and observes its own associated mode values (the name of the related collaborations). If a mode value in the array matches the mode value associated to a DMSC, it switches to the corresponding mode. However, if there is no corresponding match, the DMSC remains inactive. While it was possible to create different output mode ports on the SMC (one for each DMSC) and connect them to the DMSCs via connectors, this complicates the modeling of the RC. Thus we chose the first approach to render the diagram more compact and comprehensible.

Once the SMC and the DMSC(s) are constructed, they are placed inside a composition which we term as a *Deployment Macro Structure* (or DMS) to differentiate it from the GMS. The DMS is then placed in a repetition context termed as DRCT or *Deployment Repetition Context*. We do not concern ourselves with hierarchical or parallel DSMCs and the DSMC is sequentially executed. Hence the IRD is utilized to make the GSM associated to the DSMC equivalent to a SMA.

Another apparent change in the modeling of the DGMA is the introduction of an internal component inside the DRCT responsible for relaying the initial state of the GSM to the DSMC. In a traditional GMA, the initial state for the mode-automata (and in turn the SMC) is produced by an application component usually at a higher level of hierarchy. The application component may have a number of input event ports and an output state port. Initially some events are generated which are taken by that application component as input in order to produce as output, the initiate state. After that, the application component remains inactive due to the absence of the events arriving on its input ports.

However the nature of DGMA is different from that of GMA in the sense that at the deployment level, there is no notion of the hierarchical structure of an application. Hence it is not evident to determine which component produces the initial state at the application level and how to link that initial state to the DGMA and in turn the DSMC. For this, we create a component inside the DRCT having the *InitialStateComponent* (ISC) stereotype. This component contains only one output port of the enumerated *States* type having the shape of {1}. This provides the user defined initial state of the GSM for linking to the DSMC. Once a transition to another state occurs, the IRD allows to provide the information about the previous state and the destination state is treated as the source state for the subsequent transition of the GSM.

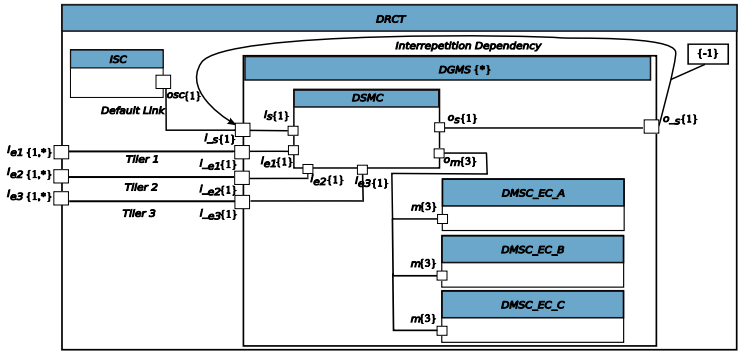

Figure 17: *An abstract overview of the DGMA*

Also we need to address the issue related to the incoming events arriving in the DGMA shown in figure 17. In a traditional GMA, the application takes the events either from the external environment (for example user generated stimuli) or the events are produced randomly in the application itself (due to a EC). However in DGMA, the incoming events have to be linked directly to the deployment level. At the RTL level, these events are basically the non deterministic user specified input by means of a UART interface into the PDR system. The user is initially given a set of options for configuration selection (these options are treated as input events) and he can choose among the different configurations modeled at the deployment level, depending upon different QoS criteria such as reconfiguration time and consumed FPGA resources.

In order to link the user specified inputs (events) to the deployment level, the DGMA has n number of event ports of shape {1,*} where n is the number of possible configurations: the first dimension on a port indicates that only one event value arrives at a particular instance of time, while the second dimension indicates a temporal dimension. These event ports are of the boolean type. The event values serve to cause a transition in the GSM depending upon the satisfaction of the expression related to a trigger, which in turn causes a transition.

It should be observed that the input event ports of the DGMA are not linked to any higher abstraction level of the application, but via model transformations, at the RTL level, are in fact treated as the input ports of the UART interface present in the Processor Sub System of the highest hierarchical entity (i.e. top.vhd) of the PDR system. This is explored more in detail later on. Figure 17 shows a complete overview of our proposed control structure.

7.3 Introducing control determinism in the RTL level

As elaborated in the precedent section, it is evident that control events are generally non deterministic in nature and depend upon the user input, while data computations in Gaspard are deterministic and operate in a regular manner. Hence there is a need to create a compatibility between the two. The notion of an *EventObserver* (EOR) is thus introduced at the RTL level in the highest hierarchical PDR system entity. We avoided adding this concept at the modeling level in order to distance the user from event arrival management which is

viewed as a lower abstraction level detail. Figure 18 show the overview of the top level entity of the PDR system.

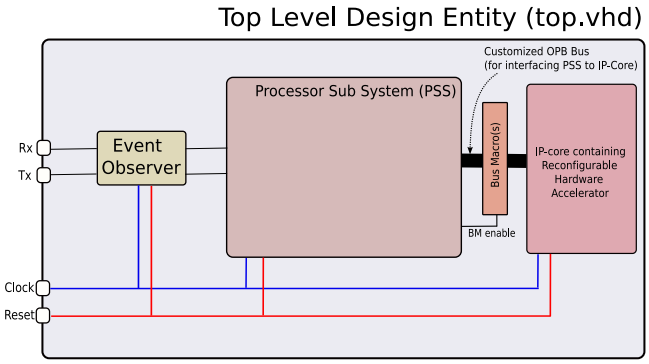

Figure 18: *A global overview of the PDR system*

This component takes user inputs at irregular time intervals and produces events at regular time intervals for regular arrival of control events into the GSM. This component has input and output event ports (*EVENTIN* and *EVENTOUT* respectively) as well as the *CLK* and *RESET* ports for clock and reset signals. The *EVENTIN* port is connected to the top level *UART_Rx* input port while the *EVENTOUT* port is connected to the *system_i*'s (instance of the processor sub module) UART_Rx input port. The algorithm related to the EOR is presented below using an informal semantic.

```
Sentivity List (CLK, EVENT)
if
  CLK is TRUE and EVENT
  then
  EVENTOUT = EVENT;
else if
  CLK = TRUE AND NOT EVENT
  then
  EVENT OUT = DEFAULT VALUE;
end if;
End Sentivity List
```

The user input can arrive irregularly at any instance of time, where as an event value is need at each instance of time t in order to respect Gaspard semantics. EOR listens on its input port, and at each rise of clock, checks if an event is present or not. In the first case, the event is sent to the processor subsystem and in turn the reconfiguration controller which causes a successful state transition (or a self transition). In the second case, if there is no user driven input event at time t, then the EOR generates a default event *event_d* which causes a self transition in the state machine. This value can be viewed as a special value among the set of values corresponding to the *all* expression, which catches any event not specified in related transitions and causes a self transition in the GSM. ξ is the set of all possible events. The overall relation is given below as:

$$E = \{e\_1, e\_2, e\_3\} \quad , \quad all = \{\xi \setminus E\} \cup \{e\_d\}$$

Figure 19 illustrates the relations between the events and the states (configurations) in a GSM in the RC. A self transition does not switch the current configuration, while a transition to a different state causes the controller to dynamically switch to the configuration corresponding to the transited destination state.

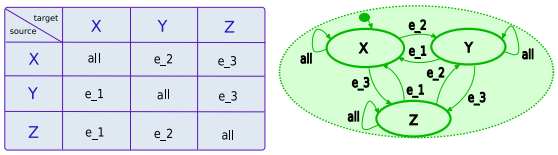

Figure 19: *Different representations of GSM: Events and state relations*

While this notion does allow to introduce regularity in the arrival of control events, it is quite possible that a control event and the eventual configuration switch causes a disrupt in the data flow of the application implemented as a RHA. It is thus critical to determine the precise moment when to effectively switch a configuration while avoiding the failure of the required application functionality. Our works could benefit from the notion of *Degree of Granularity* proposed in [30] which effectively responds to the synchronization of the control/data flow.

7.4 Extension of existing Transformation rules

It is not our objective here to fully detail the model transformations in our design flow which are considered as the underlying technical details. An initial description of the existing model transformations for the RTL level can be found in [32]. However, we do highlight some of the important new extensions that we have integrated into the RTL level.

7.4.1 Integration of the hardware design into the PDR system

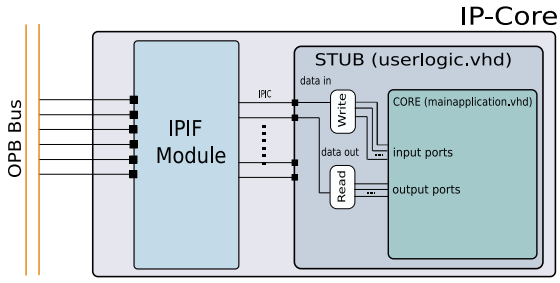

Figure 20: *An abstract overview of the IP-Core*

As elaborated earlier, the design flow in [32] creates a single hardware accelerator that is implemented in a targeted FPGA as a black box and there is no notion of heterogeneity in the final design. However, as our PDR architecture is composed of different heterogeneous communicating components (the RC, the IBM coreconnect buses, memory controllers, UART peripheral etc.), it

is essential that the generated hardware design successfully integrates into the static part of the PDR architecture. Xilinx provides the notion of an Intellectual Property-core InterFace (IPIF) module, that acts as a hardware bus wrapper specially designed to ease custom IP core interfacing with the IBM Coreconnect buses using IPIC connections. It can also be used for other purposes such as connecting the OPB bus to a DCR bus: another bus of the Coreconnect technology [27]. There exist two versions of the IPIF: a PLB IPIF for PLB attachment, and OPB IPIF, for OPB attachment [63]; [64]. A custom peripheral that connects to any of the two buses must meet the principles of the OPB/PLB protocol: matching of interface signals, for example. In our design flow, the RHA is connected with the OPB bus but can also be integrated with the PLB bus. A RHA generated via the model transformations is intended to be integrated as a slave peripheral connected to the chosen bus for communicating with the RC. However at the modeling level, the designer does not have any knowledge of the targeted PDR architecture and this is an underlying low level detail. In order to make the RHA compatible with the interface signals of the OPB and in turn the IPIF module, we create several input and output ports for different functionalities (input data, byte enable, read and write, transfer acknowledgment etc) at the top level design entity of the application (MainApplication.vhd) at the RTL model via the corresponding model transformation rules. Afterwards the IPIF wizard in the EDK can be used to generate the template for the custom peripheral containing the communication logic (IPIF module) and the inner user logic module. We treat the user logic (the userlogic.vhd) file as a stub in which the final hardware design (the *core*) is instantiated. The stub allows the bus master to read/write the output and input signals of the core respectively. Figure 20 shows the final structure of our IP-core.

Afterwards the IPIF wizard can be invoked again to import this peripheral into the PDR system resulting in a successful integration of the hardware accelerator. Currently this integration is a manual process, however model transformations can be extended to automatically generate the top level IP-Core VHDL file, an interface module and the stub module which itself contains the core submodule. This approach can be seen as a complementary approach as present in [39].

Once all the configurations of the RHA are imported as OPB peripherals (having different version names) in the EDK, the project files (***peripheral_xst.prj***) related to each version of the RHA are then modified manually as specified in the EAPR flow before eventual synthesis is carried out. Readers should refer to [57] for a detail explanation.

7.4.2 Barriers and Pipeline stages

The MoC of Gaspard authorizes the pipelined execution of an application. A data array produced by one task of the application (or architecture) can be consumed by the next successive task. In terms of a hardware accelerator, pipelined execution of tasks permit to increase the operational frequency while decomposing its critical path. For that this execution is really pipelined, registers having the same clock rate are introduced in different data paths. *Tilers* containing registers in their data paths have been introduced in [32], they allow to generate a flow of arrays in the data paths of task parallelism. This flow of arrays implies that several tasks are executed at the same time but each iterates on a different

instance on the temporal dimension in their repetition space. However, this pipeline introduces a latency in the production of output arrays of a task as it is necessary to fill the pipeline stages of tasks.

However, the limitations related to the utilization of the pipelined approach in task parallelism are evident. A de-synchronization can occur when the data dependency does not fills all the stages of a pipeline as shown in figure 21. It is thus up to the designer of the application to guarantee that that use of a task not does de-synchronize the computations.

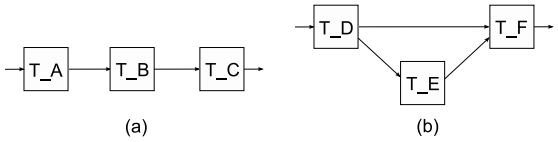

Figure 21: *A representation of de-synchronization in task parallelism*

In order to avoid the de-synchronization of our hardware accelerator(s), we have implemented a model transformation rule which aims to create *Barriers* and *Pipeline Stages* for each compound component of the modeled application. A pipeline stage corresponds to an actual pipeline stage in the application and can be composed of several sub components. The barrier insures that the for the next successive pipeline stage, the input values from the previous stage are present and that data dependency fills all the pipeline stages. This model transformation is implemented as a black box written in Java which is called from typical MoMOTE transformation rule. The algorithm of this transformation is described in a non formal semantic.

```
CreateBarriersandPipelineStages(CompoundComponent hwcc)
{
  //list of input port type
  List <Port>InputPortList = new ArrayList();
  for (port hwp : hwcc.getPorts())
    if (hwp instanceof InputPort)
    {
      InputPortList.add(hwp);
    }

  //list of connector type
  List <Connector>listconnector = new ArrayList();
  for (InputPort hwip : InputPortList)
  {
   listconnector.addall(hwip.getconnect());
  }

  //list of component instance type
  List <ComponentInstance>CIToschedule = new ArrayList();
  CIToschedule.addall(hwcc.getComponentInstance());

  int count = 1;
  do
  {
  create PipelineStage ps;
  ps.setName("No"+count);
  ps.setPosition(count);
  hwcc.add(ps);
```

```
  create Frontier frontier;
  hwcc.add(frontier);

  create PipelineStage(hwcc, listconnector, CIToschedule, ps, frontier);
  count++;

  }
  while (!(CIToschedule.isEmpty()));
}
```

The rule is composed of several parts (functions). The first part of the rule initially determines if the compound component (CC) in question contains input ports. If this is true, then the component is considered as an internal CC of the main application other wise it is the main application component itself (as at the UML modeling level, the top level main application entity does not have any input or output ports). For an internal CC, the presence of input port(s) determine the presence of data arrival. Afterwards we determine the connectors connecting to these input ports and the number of all component instances present in the CC. A Pipeline stage and a barrier are then created repeatedly until the list of the component instances in the CC is not empty. This function calls the second part of the rule.

```
CreatePipelineStage(hwcc, listconnector, CIToschedule, ps, frontier)
{
  //list of component instance type
  List <ComponentInstance>locallist = new ArrayList();
  locallist.add(CIToschedule);

  //list of connector type
  List <Connector>connectorfornextstage = new ArrayList();
  connectorfornextstage.addall(listconnector);

  for (ComponentInstance hwci : locallist)
  {
    //list of port instance type
    List <Port>hwpilist = new ArrayList();
    hwpi = getPI(hwci);

    //list of port instance type
    List <Port>locallist = new ArrayList();
    locallist.addall(hwpilist);

    for (PortInstance hwpitemp : locallist)
    {
      if (listconnector.contains(hwpitemp.getconnect().get(0)))
      {hwpilist.remove(hwpitemp);}
    }

    if (hwpilist.isEmpty())
    {
      ps.getComponentInstance().add(hwci);
      CIToschedule.remove(hwci);
      // list of connector type
      List <Connector>hwc = new ArrayList();
      hwc = getC(hwci);
      frontier.getconnectors.addall(hwc);
      connectorfornextstage.addall(hwc);
    }
  }
 listconnector.addall(conenctorfornextstage);
}
```

The second part stores the list of component instances to be placed in the pipeline stages as well as the input connectors. For each component instance *hwci* we determine the number of input ports (excluding the clk and reset ports for that component) via the function *getPI* (not described here). This collection of input ports is thus stored in a list. For each port instance of a component instance, we determine if the connector connecting to this port instance is present in the initial list of connectors connected to the CC itself. If so, this port instance is considered active and removed from the list. When the list pertaining to all the port instances of a component instance are empty, this component instance is added in the pipeline stage *ps* and removed from the list of component instances of the CC. The connectors connected to the outport ports of this component instance are then calculated via the *getC* function (not described here) and added to the barrier *frontier*. These connectors also replace the initial information related to the connectors and serve as input connectors for the next pipeline stage. Figure 22 shows an illustration of the aforementioned rule.

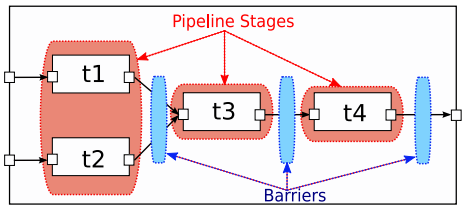

Figure 22: *A representation of the pipeline stages and barriers*

7.4.3 Model Transformations : From MoMOTE to QVTO

As explained in the section related to Gaspard, due to the arrival of new evolved transformation tools which can support the complex model transformations present in our framework, we have decided to rewrite the exiting GASPARD2RTL MoMOTE based transformation rules in QVTO. As the gap between the concepts in MARTE and Gaspard metamodels is not large enough, the significant but small details can be elevated to the MARTE level. This will allow us to move directly from the MARTE metamodel to the RTL metamodel and eventual code generation. This results in several advantages. The first results in the decrease in the number of transformation rules (as the MARTE2GASPARD model transformation and the Gaspard metamodel is no longer needed). This also allows us to easily maintain the corresponding transformation changes due to any changes in the MARTE standard and its metamodel. Thus the design flow is standardized according to MARTE specifications. This is currently a work in progress. It should be mentioned that the migration to QVTO does not effect the generated code and the subsequent implementation results.

8 Future Works and Perspectives

Currently our design flow does not manages the loss of the data flow between the configuration switching, and can inspire from [30]. At the moment, the final placement of PRR and bus macros is determined by the designer; it is a tedious

task and can effect the overall reconfiguration time. Works such as [47], [11] could complement our design flow and help the designer in automating the PDR flow. Solutions such as presented in [38], can resolve the issues related to area constraints for static/dynamic regions. At present, the modeled application is placed as a single hardware accelerator. However, some key kernels of the application could be implemented as hardware accelerators while other parts can be executed in the RC. This adds an additional layer of complexity of the synchronization of the control/data flow and the communication in the overall system. Bitstream relocation between two PRRs is also an interesting perspective as presented in [11], [5]. At the moment, the architecture of the PDR system is absent at the high modeling level. An initial methodology was introduced in [46] but model transformations are currently absent. At the RTL level, the control mode automata is directly converted as a state machine code for one of the PowerPCs available in the XUP board. In case of an architecture modeled at UML supporting multiple processors, at the allocation level, the correct processor must be linked to the control concept modeled at deployment. Thus a link must be created to relate the two levels for a correct allocation. A good comprise could be to introduce the notion of control at each of the modeling levels (application, architecture, association and deployment).

9 Conclusion

This paper presents a novel model driven methodology to move from high level MARTE specifications to reconfigurable architectures such as FPGAs, and specifically those supporting partial dynamic reconfiguration (PDR). Our methodology allows to specify complex intensive signal processing application such as a multimedia codecs and digital filters in a graphical language, which via model transformations, are implemented as hardware functionalities in a targeted FPGA. These functionalities retain the inherent task and data parallelism specified at the modeling level. Extensions have been made to the existing control/data flow concepts and the deployment level in our framework to integrate the PDR aspects. A deterministic control approach has also been proposed to integrate non regular control events in Gaspard. An initial version of generating a complete IP core from model transformations has also been presented, along with a solution to avoid de-synchronization related to task parallelism in the modeled applications. The IP-core can be successfully integrated into a PDR architecture in order to build a complete system. Finally a case study is illustrated to validate our MDE based design flow. Using MDE and model transformations, we can produce as output, the source code for the reconfiguration controller and the dynamically reconfigurable module. The code can then be used as input for commercial tools for final FPGA synthesis. Currently we adhere to the Xilinx based PDR design flow due to its availability and extensible nature. However our PDR based methodology can be used as a template in order to implement other existing or future PDR based fine grain reconfigurable architectures. A potential extension could be to target coarse grain reconfigurable architectures supporting multiple FPGAs.

10 Acknowledgements

We would like to thank Huafeng Yu and Abdoulaye Gamatié for their valuable inputs regarding our propositions; and Juanjo Noguera at Xilinx Research Labs for responses related to the initial implementation of dynamic reconfiguration.

References

[1] Atat, Y., and Zergainoh, N. Simulink-based MPSoC Design: New Approach to Bridge the Gap between Algorithm and Architecture Design. In *ISVLSI'07*, pages 9–14, 2007.

[2] Atitallah et al. Multilevel MPSoC simulation using an MDE approach. In *SoCC 2007*, 2007.

[3] Bayar, S., and Yurdakul, A. Dynamic Partial Self-Reconfiguration on Spartan-III FPGAs via a Parallel Configuration Access Port (PCAP). In *2nd HiPEAC workshop on Reconfigurable Computing, HiPEAC 08*, 2008.

[4] Becker et al. Real-Time Dynamically Run-Time Reconfigurations for Power/Cost-optimized Virtex FPGA Realizations. In *VLSI'03*, 2003.

[5] Becker et al. Enchancing Relocability of Partial Bit-streams for Run-Time Reconfiguration. *Field-Programmable Custom Computing Machines, FCCM 2007*, pages 35–44, 2007.

[6] Berthelot et al. A Flexible system level design methodology targeting run-time reconfigurable FPGAs. *EURASIP Journal of Embedded Systems*, 8(3):1–18, 2008.

[7] Blodget et al. A lightweight approach for embedded reconfiguration of FPGAs. In *Design, Automation & Test in Europe, DATE'03*, 2003.

[8] Boden et al. GePARD - a High-Level Generation Flow for Partially Reconfigurable Designs. In *ISVLSI 2008*, 2008.

[9] Boulet, P. Array-OL Revisited, Multidimensional Intensive Signal Processing Specification. Technical report, INRIA, 2007. `http://hal.inria.fr/inria-00128840/en/`.

[10] Canto et al. Self Reconfiguration of Embedded Systems Mapped on Spartan-3. *ReCoSoC'08*, 2008.

[11] Carver et al. Relocation and Automatic Floor-planning of FPGA Partial Configuration Bit-Streams. Technical report, Microsoft Research, 2008.

[12] Cesario et al. Component-Based Design Approach for Multicore SoCs. *Design Automatic Conference, DAC'02*, 00:789, 2002.

[13] Claus et al. A new framework to accelerate Virtex-II Pro dynamic partial self-reconfiguration. *IPDPS 2007*, pages 1–7, 2007.

[14] Cuoccio et al. A generation flow for self-reconfiguration controllers customization. *Forth IEEE International Symposium on Electronic Design, Test and Applications, DELTA 2008*, pages 279–284, 2008.

[15] Damasevicius, R., and Stuikys, V. Application of UML for hardware design based on design process model. In *ASP-DAC'04*, 2004.

[16] DaRT team. GASPARD SoC Framework, 2009. `http://www.lifl.fr/DaRT`.

[17] Dorairaj et al. PlanAhead Software as a Platform for Partial Reconfiguration. *Xcell Journal*, 55:68–71, 2005.

[18] Eclipse. Eclipse Modeling Framework. `http://www.eclipse.org/emf`.

[19] Eclipse. Eclipse Modeling Framework Technology. `http://www.eclipse.org/emft`.

[20] Eclipse. EMFT JET. `http://www.eclipse.org/emft/projects/jet`.

[21] Gailliard et al. Transaction level modelling of SCA compliant software defined radio waveforms and platforms PIM/PSM. In *Design, Automation & Test in Europe, DATE'07*, 2007.

[22] Gamatié et al. A Model for the Mixed-Design of Data-Intensive and Control-Oriented Embedded Systems. Research Report RR-6589, INRIA, 2008. `http://hal.inria.fr/inria-00293909/fr`.

[23] Gamatié et al. A model driven design framework for high performance embedded systems. Research Report RR-6614, INRIA, 2008. `http://hal.inria.fr/inria-00311115/en`.

[24] Guo et al. Optimized generation of data-path from C codes for fpgas. In *Design, Automation & Test in Europe, DATE'05*, pages 112–117, 2005.

[25] Huebner et al. Elementary Block Based 2-Dimensional Dynamic and Partial Reconfiguration for Virtex-II FPGAs. In *IPDPS 2006*, 2006.

[26] Huebner et al. New 2-Dimensional Partial Dynamic Reconfiguration Techniques for Real-Time Adaptive Microelectronic Circuits. In *ISVLSI'06*, 2006.

[27] IBM Corporation. The Coreconnect Bus Architecture, white paper. In *International Business Machines Corporation*, 2004.

[28] Koch et al. An adaptive system-on-chip for network applications. In *IPDPS 2006*, 2006.

[29] Koudri et al. Using MARTE in the MOPCOM SoC/SoPC Co-Methodology. In *MARTE Workshop at DATE'08*, 2008.

[30] O Labbani. *Modélisation à haut niveau du contrôle dans des applications de traitement systématique à parallélisme massif*. PhD thesis, LIFL / USTL, France, 2006.

[31] Le Beux et al. A Model Driven Engineering Design Flow to generate VHDL. In *International ModEasy'07 Workshop*, 2007.

[32] Le Beux, S. *Un flot de conception pour applications de traitement du signal systématique implémentées sur FPGA à base d'Ingénierie Dirigée par les Modeles*. PhD thesis, LIFL / USTL, France, 2007.

[33] Lysaght et al. Invited Paper: Enhanced Architectures, Design Methodologies and CAD Tools for Dynamic Reconfiguration of Xilinx FPGAs. In *International Conference on Field Programmable Logic and Applications, FPL'06*, 2006.

[34] F. Maraninchi and Y Rémond. Mode-automata: a new domain-specific construct for the development of safe critical systems. *Sci. Comput. Program.*, 46(3):219–254, 2003.

[35] McUmber et al. UML-based analysis of embedded systems using a mapping to VHDL. In *IEEE International Symposium on High Assurance Software Engineering, HASE'99*, pages 56–63, 1999.

[36] Mens, T., and Van Gorp, P. A taxonomy of model transformation. In *Proceedings of the International Workshop on Graph and Model Transformation, GraMoT 2005*, volume 152, pages 125–142, 2006.

[37] Mohanty et al. Rapid design space exploration of heterogeneous embedded systems using symbolic search and multi-granular simulation. In *LCTES/Scopes 2002*, 2002.

[38] Montone et al. A design workflow for the identification of area constraints in dynamic reconfigurable systems. *4th IEEE International Symposium on Electronic Design, Test and Applications, DELTA 2008*, pages 450–453, 2008.

[39] Murgida et al. Fast IP-Core generation in a Partial Dynamic Reconfigurable workflow. In *VLSI-SoC 2006*, 2006.

[40] OMG. MOF Query /Views/Transformations, Nov. 2005. `http://www.omg.org/cgi-bin/doc?ptc/2005-11-01`.

[41] OMG. OMG MARTE Standard. 2007. `http://www.omgmarte.org`.

[42] OMG. OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2, 2007. `http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF/`.

[43] OSI. SystemC. 2007. `http://www.systemc.org/`.

[44] Paulsson et al. Implementation of a Virtual Internal Configuration Access Port (JCAP) for Enabling Partial Self-Reconfiguration on Xilinx Spartan III FPGAs. *International Conference on Field Programmable Logic and Applications, FPL 2007*, pages 351–356, 2007.

[45] Planet MDE. Portal of the Model Driven Engineering Community. 2007. `http://www.planetmde.org`.

[46] Quadri et al. A Model Driven design flow for FPGAs supporting Partial Reconfiguration. Hindawi Publishing Corporation, 2009. To Appear.

[47] Scholz, R. Adapting and Automating XILINX Partial Reconfiguration Flow for Multiple Module Implementations. In *Book Chapter in Reconfigurable Computing : Architectures, Tools and Applications*, 2007.

[48] Schuck et al. A framework for dynamic 2D placement on FPGAs. In *IPDPS 2008*, 2008.

[49] Schuck et al. An Interfance for a Decentralized 2D-Reconfiguration on Xilinx Virtex-FPGAs for Organic Computing. In *ReCoSoC'08*, 2008.

[50] Sedcole et al. Modular Partial Reconfiguration in Virtex FPGAs. In *International Conference on Field Programmable Logic and Applications, FPL'05*, pages 211–216, 2005.

[51] S. Sendall and W Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Software*, 20(5):42–45, 2003.

[52] Stevens, P. A landscape of bidirectional model transformations. In *Generative and Transformational Techniques in Software Engineering 2007, GTTSE'07*, 2007.

[53] Tumeo et al. A Self-Reconfigurable Implementation of the JPEG Encoder. *ASAP 2007*, pages 24–29, 2007.

[54] Xilinx. Two Flows for Partial Reconfiguration: Module Based or Difference Based. In *Xilinx Application Note XAPP290, Version 1.1*, 2003.

[55] Xilinx. Two Flows for Partial Reconfiguration: Module Based or Difference Based. In *Xilinx Application Note XAPP290, Version 1.2*, 2004.

[56] Xilinx. Early Access Partial Reconfigurable Flow. 2006. `http://www.xilinx.com/support/prealounge/protected/index.htm`.

[57] Xilinx. Early Access Partial Reconfigurable User Guide for ISE 8.1.01i. 2006. `http://www.xilinx.com/support/prealounge/protected/index.htm`.

[58] Xilinx. Early Access Partial Reconfigurable User Guide for ISE 9.2.04i. 2008. `http://www.xilinx.com/support/prealounge/protected/index.htm`.

[59] Xilinx. Embedded Development Kit (EDK) Software. 2009. `http://www.xilinx.com/ise/embedded_design_prod/platform_studio.htm`.

[60] Xilinx. Fast Simplex Link Channel (FSL). 2009. `http://www.xilinx.com/products/ipcenter/FSL.htm`.

[61] Xilinx. ISE Foundation Software. 2009. `http://www.xilinx.com/ise/logic_design_prod/foundation.htm`.

[62] Xilinx. OPB hwicap product specification. Technical report, 2009.

[63] Xilinx. OPB IPIF Architecture. 2009. `http://www.xilinx.com/products/ipcenter/OPB_IPIF_Architecture.htm`.

[64] Xilinx. PLB IPIF Architecture. 2009. `http://www.xilinx.com/products/ipcenter/plb_ipif.htm`.

[65] Xilinx. Virtex-VI product page. 2009. `http://www.xilinx.com/products/v6s6.htm`.

[66] Xilinx. XUP-V2Pro Board, 2009. `http://www.xilinx.com/univ/xupv2p.html`.

[67] Yu, H. *A MARTE based reactive model for data-parallel intensive processing: Transformation toward the synchronous model*. PhD thesis, LIFL / USTL, France, 2008.

Centre de recherche INRIA Lille – Nord Europe
Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399